Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem

Paul W.P.J. Grefen
Computer Science Department
University of Twente
grefen@cs.utwente.nl

Abstract

Integrity control is generally considered an important topic in the field of database system research. In the database literature, many proposals for integrity control mechanisms can be found. A large group of proposals has a formal character, and does not cover complete algorithms that can be used in a real-world database system with multi-update transactions. Another group of proposals is system-oriented and often lacks a complete formal background on transactions and integrity control; algorithms are usually described in system terms. This paper combines the essentials of both groups: it presents a declarative specification of a transaction-based integrity control technique that has a solid formal basis and can easily be applied in real-world database systems. The technique, called *transaction modification*, features simple semantics, full transaction support, and extensibility to parallel data processing. These claims are supported by a prototype implementation of a transaction modification subsystem in the high-performance PRISMA/DB database system. This paper shows that it is well possible for an integrity control technique to combine a formal approach with complete functionality and high performance.

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

1 Introduction

Integrity control is generally considered an important topic in the field of database systems research and practice. Many proposals for integrity control mechanisms can be found in database literature [10]. Often, a proposal can be classified in one of two groups described below. One group of proposals has a formal character, and does not cover complete algorithms that can be used in a real-world database system context with arbitrary multi-update transactions. The work presented in [2] is an example: it only deals with single-tuple insert and delete operations. The algorithms presented in [14] only handle single tuple operations and transactions in which the order of operations is immaterial. Another group of proposals is system-oriented and often lacks a complete formal background on transactions and integrity control; algorithms are usually described in system terms. A well-known example is the work on query modification described in [19]: this work is closely connected to the QUEL language and does not take transactions into account. The work in [17] informally describes integrity control algorithms in the context of the SABRE systems. In active database systems like POSTGRES [20], integrity control algorithms rely on the complex semantics of the underlying production rule system.

This paper proposes a technique for integrity control with a clear formal basis that can easily be applied in real-world database systems. The technique is called *transaction modification*, and, as suggested by its name, is based on the transaction concept [6]. Transaction modification modifies an arbitrary transaction that may violate the integrity of a database, such that the execution of the modified transaction is guaranteed to be correct. As shown in the sequel of this paper, the algorithms for this technique can be specified in a clear declarative

fashion, providing a solid basis for the implementation of the technique. This claim is supported by a prototype implementation of a transaction modification subsystem in the parallel main-memory PRISMA/DB system [1]. The implementation further demonstrates the usability of the technique in distributed and high-performance environments [7, 9].

The structure of this paper is as follows. Section 2 discusses a number of basic database concepts necessary for the sequel of the paper. Section 3 defines the notion of integrity constraints and its relation to databases and transactions executed against databases. The specification of integrity constraints is discussed in Section 4. Section 5 describes the transaction modification technique for integrity control on a declarative level. Operational aspects of the technique are discussed in Section 6. The paper ends with a number of conclusions and a few remarks on the performance of the prototype implementation.

2 Basic concepts

This section discusses the basic concepts in the field of relational databases supplying the formal background for the sequel of the paper. First, the database structures are defined. Next, operations on the databases in the form of transactions are discussed. The transaction concept plays a central role in the description of database integrity and integrity control in the sections that follow.

2.1 Databases

The relational database structures consist of relation schemas and instances, and database schemas and instances. Database transitions are also discussed here, because they are an important ingredient in the sequel of the paper.

Definition 2.1. A *relation schema* $\mathcal{R}$ consists of a relation name and a list of attributes $\langle A_1, \cdots, A_n \rangle$. Each attribute A_i is defined on a domain $dom(A_i)$. The type of $\mathcal{R}$ is defined as $dom(\mathcal{R}) = dom(A_1) \times \cdots \times dom(A_n)$. A *relation* or *relation state* R of relation schema $\mathcal{R}$ consists of the name of $\mathcal{R}$ and a set of elements (tuples) in $dom(\mathcal{R})$.

Definition 2.2. A *database schema* $\mathcal{D}$ is a set of relation schemas $\{\mathcal{R}_1, \cdots, \mathcal{R}_n\}$. A *database* or *database state* D of database schema $\mathcal{D}$ is a set of relation instances $\{R_1, \cdots, R_n\}$. The set of all possible database instances of schema $\mathcal{D}$ is called the *database universe*.

Definition 2.3. A *database transition* of database schema $\mathcal{D}$ is an ordered pair of database states $\langle D^{t_1}, D^{t_2} \rangle$ of schema $\mathcal{D}$, with $t_1, t_2 \in \mathbb{N}$ and $t_1 < t_2$. The values t_1 and t_2 are called the *logical times* of the database states.

Usually, a database transition describes two successive states of the database, so $t_2 = t_1 + 1$ in the definition above. This type of transition is called a *single-step transition*. The term transition is used for single-step transitions in this paper.

2.2 Transactions

Operations executed against a database are grouped into transactions to form database programs with certain characteristics. The transaction concept as defined below plays a central role in the sequel of this paper. Transactions are based on extended relational algebra programs as defined below.

Definition 2.4. An *extended relational algebra program* P is a sequence of extended relational algebra statements:

$$P = a_1; a_2; \ldots; a_n$$

The extended relational algebra extends the standard relational algebra (see e.g. [13]) with statements that enable the operational specification of actions against a database [8]. Extended relational algebra statements include assignments, insert, delete, and update statements. The symbol P_ϵ denotes the empty program.

Definition 2.5. A *transaction* T consists of an extended relational algebra program $a_1; \cdots; a_n$ enclosed in *transaction brackets*, to be executed against a database D:

$$T = (a_1; a_2; \ldots; a_n)$$

The parentheses denote the transaction brackets, respectively *begin* and *end*. During the execution of the actions a_i, the database is in a number of *intermediate states*. These states are not normal database states as they may contain temporary relations (intermediate results). If the logical time of D is t, then the state after the execution of action a_i is denoted as $D^{t.i}$. The *end* bracket takes care of the transition from $D^{t.n}$ to a normal database state: if the transaction can *commit*, temporary relations are removed from $D^{t.n}$ and the result, denoted as $\lfloor D^{t.n} \rfloor$, is installed as D^{t+1}; if the transaction must *abort*, D^t is installed as D^{t+1}. The states $D^{t.1}, \ldots, D^{t.n}$ have no semantics beyond the execution of T. The pre-transaction state D^t and post-transaction state D^{t+1} are visible to other transactions as well.

Informally, a transaction is a unit of work executed

against a database state. Speaking more formally, a transaction T can be seen as an operator that transforms a database state D into another state $T(D)$ [6], and can thus be associated with a single-step transition of a database:

$$D \xrightarrow{T} T(D)$$

According to the basic transaction model, the execution of a transaction T must satisfy the properties of atomicity, serializability, durability, and correctness. Only the atomicity and correctness properties are of interest in this paper. The first is described below, the latter is discussed in the next section.

The execution of T must always satisfy the atomicity property; this means that the effect of any execution of T on the initial database state D must be such that either the effects of T are completed fully, or D remains unchanged. So, if $T = (a_1, \ldots, a_n)$, the following must hold:

$$(T(D) = \lfloor D^{t,n} \rfloor) \vee (T(D) = D)$$

3 Integrity constraints

This section presents a short treatment on the formal background of integrity constraints. Constraints are first discussed in the static context of databases, and thereafter in the dynamic context of transactions. The specification of constraints is discussed in Section 4.

3.1 Databases and integrity constraints

Below, the concept of *integrity constraint* is defined. In this definition, constraints are divided into *state constraints* that describe properties of *correct database states*, and *transition constraints* that describe properties of *correct database transitions*.

Definition 3.1. Let $\mathcal{D}$ be a database schema. A *state constraint* I^s is a boolean function that is evaluated over a database state D from the database universe $U_{\mathcal{D}}$ defined on $\mathcal{D}$:

$$I^s : U_{\mathcal{D}} \rightarrow bool$$

Definition 3.2. A *correct database state* $D \in U_{\mathcal{D}}$ satisfies each element of a set of state constraints $\mathcal{I}^s = \{I_1^s, \cdots, I_m^s\}$ defined on $\mathcal{D}$. The set of correct database states with schema $\mathcal{D}$ and constraint set $\mathcal{I}^s$ is denoted as:

$$U_{\mathcal{D}}^{\mathcal{I}^s} = \left\{ D \in U_{\mathcal{D}} \;\middle|\; \bigwedge_{i=1}^{i=m} I_i^s(D) \right\}$$

A state constraint describes the static properties of a database, i.e. the properties that a database should satisfy at one given moment.

Definition 3.3. Let $\mathcal{D}$ be a database schema. A *transition constraint* I^t is a boolean function that is evaluated over a pair of database states or database transition $\langle D_1, D_2 \rangle$ defined on $\mathcal{D}$:

$$I^t : U_{\mathcal{D}} \times U_{\mathcal{D}} \rightarrow bool$$

Definition 3.4. A *correct database transition* $\langle D_1, D_2 \rangle$ defined on schema $\mathcal{D}$ satisfies each element of a set of transition constraints $\mathcal{I}^t = \{I_1^t, \cdots, I_n^t\}$ defined on $\mathcal{D}$. The set of correct database transitions with schema $\mathcal{D}$ and constraint set $\mathcal{I}^t$ is denoted as:

$$V_{\mathcal{D}}^{\mathcal{I}^t} = \left\{ \langle D_1, D_2 \rangle \in U_{\mathcal{D}} \times U_{\mathcal{D}} \;\middle|\; \bigwedge_{i=1}^{i=n} I_i^t(D_1, D_2) \right\}$$

A transition constraint describes the correct transitions of a database; as such, it describes dynamic properties of a database. Therefore, transition constraints are also referred to as *dynamic constraints*.

3.2 Transactions and integrity constraints

The integrity of the database in terms of integrity constraints as defined above has its effect on the set of transactions that are allowed to be executed and successfully commited against a database: the execution of a transaction may not violate the integrity of a database. Note that the transaction model discussed above, in which intermediate database states have no semantics beyond the execution of a transaction, implies that integrity is only defined with respect to pre- and post-transaction database states, not to intermediate states.

The fact that each database state has to satisfy all state constraints defined on the database means that the execution of a transaction T on a correct state D may never result in a database state that violates any constraint in $\mathcal{I}^s$; so the following should hold:

$$\left(\bigwedge_{i=1}^{i=m} I_i^s(D) \right) \Rightarrow \left(\bigwedge_{i=1}^{i=m} I_i^s(T(D)) \right)$$

The fact that each database transition has to satisfy all transition constraints means that given a correct database state, the execution of a transaction T may never imply a transition that violates any constraint in $\mathcal{I}^t$; so the following should always hold:

$$\left(\bigwedge_{i=1}^{i=m} I_i^s(D) \right) \Rightarrow \left(\bigwedge_{i=1}^{i=n} I_i^t(D, T(D)) \right)$$

Given these obeservations, we can define the correctness of a transaction as follows.

Definition 3.5. A transaction T is *correct* with respect to a correct database state D and a set of integrity constraints $\mathcal{I}$ if and only if a commited execution of T on D does not imply a database transition that violates any transition constraint in $\mathcal{I}$, and the post-transaction database state $T(D)$ does not violate any state constraint in $\mathcal{I}$. A transaction that does not comply with this requirement is called *incorrect*.

4 Integrity specification

In the previous section, the concept of integrity constraints was introduced. This section discusses the specification of integrity requirements, i.e. a formalism in which the integrity of a database can be expressed. To come to a flexible framework, it is important to distinguish two forms of specification:

Integrity Constraint As discussed before, an integrity constraint specifies a condition that must be met by a database (transition), but includes no specification of how the integrity must be controlled. As such, it is a purely declarative specification of integrity requirements on a database.

Integrity Rule To come to a flexible basis for integrity control, a more operational form for integrity control is required, giving information about how to enforce integrity. This form is called *integrity rule*.

If integrity control is to be performed in a default way (e.g. all incorrect transactions are to be aborted), the specification of integrity constraints is sufficient and rules can be derived automatically. If more flexibility is required, rules have to be specified by the database designer. The database system can be of help, however, by generating parts of rules as shown in the sequel of this paper.

4.1 Integrity constraints

A language for the specification of integrity constraints is defined below. The language is called $\mathcal{CL}$ and consists of a set of symbols, terms, atomic formulas, and well-formed formulas. It is based on the tuple relational calculus [21].

Definition 4.1. The *alphabet* for the specification of integrity constraints consists of the following symbols:

- The set of value constants $C = \{c_1, c_2, \ldots\}$, the set of tuple set constants $M = \{R, S, \ldots\}$, and the set of tuple variables $V = \{x, y, z, \ldots\}$.
- The set of tuple function symbols $FT = \{.\}$ of type $V \times C \rightarrow C$, the set of value function symbols $FV = \{+, -, *, /\}$ of type $C \times C \rightarrow C$, the set of aggregate function symbols $FA = \{SUM, AVG, MIN, MAX\}$ of type $M \times C \rightarrow C$, and the set of counting function symbols $FC = \{CNT\}$ of type $M \rightarrow C$.
- The set of value predicate symbols: $PV = \{<, \leq, =, \neq, \geq, >\}$ of type $C \times C$, the set of set predicate symbols $PM = \{\in\}$ of type $V \times M$, and the set of tuple predicate symbols $PT = \{=\}$ of type $V \times V$.
- The sets of unary connectives $CU = \{\neg\}$ and binary connectives $CB = \{\vee, \wedge, \Rightarrow\}$, the set of quantifiers $Q = \{\exists, \forall\}$, and the set of punctuation symbols $IP = \{(,)\}$.

The tuple set constants from the set M correspond with the base and auxiliary relations of a database. The base relations are the permanent relations containing the data actually stored in the database. The auxiliary relations are calculated from the base relations automatically by the database management system for specific integrity control purposes. An important type of auxiliary relation is the pre-transaction state of a relation, necessary for the specification of transition constraints.

Definition 4.2. The elements of the set of *terms* $\mathcal{T}$ are the following:

- Value constants from the set C.
- Attribute selections $x.i$, where $x \in V$ and i an integer constant from C.
- Arithmetic function applications $t_1 \vartheta t_2$, where $\vartheta \in FV$, $t_1 \in \mathcal{T}$, and $t_2 \in \mathcal{T}$.
- Aggregate function applications $\Gamma(R, i)$, with $\Gamma \in FA$, $R \in M$, and i an integer constant from C; counting function applications $\Gamma(R)$, with $\Gamma \in FC$ and $R \in M$.

Definition 4.3. The elements of the set of *atomic formulas* $\mathcal{A}$ are the following:

- Arithmetic comparisons $T_1 \vartheta T_2$, with $\vartheta \in PV$, and $T_1, T_2 \in \mathcal{T}$.
- Set membership expressions $x \in R$, where $x \in V$, and $R \in M$.

- Tuple value comparisons $x = y$, where $x \in V$ and $y \in V$.

Definition 4.4. The elements of the set of *well-formed formulas* $\mathcal{W}$ are the following:

- Atomic formulas A.
- Negations $\neg W$, with $W \in \mathcal{W}$; connections $W_1 \vartheta W_2$, with $W_1, W_2 \in \mathcal{W}$, and $\vartheta \in CB$.
- Quantifications $(\vartheta x)W$, with $\vartheta \in Q$, $x \in V$, and $W \in \mathcal{W}$.

Some examples of constraint specifications in $\mathcal{CL}$ are presented below.

Example 4.1. We take a simple beer database with two relations *beer(name,type,brewery,alcohol)* and *brewery(name,city,country)* as an example. A domain constraint I_1 and a referential integrity constraint I_2 can be specified as follows on this database:

$$I_1 : (\forall x)(x \in beer \Rightarrow x.alcohol \geq 0)$$
$$I_2 : (\forall x)(x \in beer \Rightarrow (\exists y)(y \in brewery \wedge x.brewery = y.name))$$

4.2 Integrity rules

As described above, an integrity rule is an operational form of an integrity constraint. For integrity control purposes, two types of information have to be added to an integrity constraint: information about when to enforce the constraint, and information about what to do when the constraint is violated by the database. The first type of information is provided by a *trigger set*, a set of update types that may violate the constraint. The second type of information is provided by the *violation response action*, an extended relational algebra program describing the actions to be performed upon constraint violation. Below, the formalism for specifying integrity rules is defined.

Definition 4.5. The set of *trigger specifications* on database schema $\mathcal{D}$ is defined as follows. Let U denote an elementary update type, such that $U \in \{INS, DEL\}$, and let R denote a relation name in $\mathcal{D}$. Then the set of trigger specifications on $\mathcal{D}$ consists of all possible combinations $U(R)$.

In trigger specifications, an update operation on a relation is specified as a combination of a delete and an insert operation. Triggers are combined into *trigger sets* as described below:

Definition 4.6. The set of *trigger set specifications* on database schema $\mathcal{D}$ is defined as follows. Let t be a trigger specification on schema $\mathcal{D}$ and s a trigger set specification on $\mathcal{D}$. Then the following lists are trigger set specifications:

- The single element trigger set t.
- The composed trigger set s, t.

Now the integrity rule specification language $\mathcal{RL}$ can be defined as follows:

Definition 4.7. The set of integrity rule specifications in language $\mathcal{RL}$ is defined as follows. Let ts be a trigger set specification, c an integrity constraint specification in $\mathcal{CL}$, and p an extended relational algebra program. Then the following construct is an integrity rule specification:

WHEN ts
IF NOT c
THEN p

Example 4.2. The following integrity rules in $\mathcal{RL}$ are based on domain and referential integrity constraints I_1 resp. I_2 introduced before:

R_1 : WHEN $INS(beer)$
IF NOT $(\forall x)(x \in beer \Rightarrow x.alcohol \geq 0)$
THEN *abort*

R_2 : WHEN $INS(beer), DEL(brewery)$
IF NOT $(\forall x)(x \in beer \Rightarrow (\exists y)(y \in brewery \wedge x.brewery = y.name))$
THEN $temp = \pi_{brewery}beer - \pi_{name}brewery;$
$insert(brewery, \pi_{name,null,null}temp)$

As may be clear from the specification, rule R_1 specifies an aborting approach to constraint violation handling (incorrect transactions are aborted), whereas R_2 specifies a compensating violation response action (incorrect updates are compensated by additional updates).

5 Integrity control

As stated in the introduction, the main topic of this paper is a declarative specification of an integrity control technique that has a solid formal background and easy application in practice. The *transaction modification* technique to be discussed in this section complies with both requirements. The technique is based on the formal concepts of the previous sections, and is specified in a clear top-down, declarative fashion. As suggested by its name, the technique is strongly transaction-oriented, providing the necessary functionality for practice.

In this section, the basic transaction modification technique is discussed. To improve the applicability in

practice, a few additions can be made. These additions are treated in Section 6.

5.1 Transaction modification

Arbitrary transactions submitted to a database system by a user or application may violate the integrity of the database. The integrity control subsystem of a DBMS should maintain the integrity of the database, regardless of the actions of these transactions. In the transaction modification approach to integrity control, a possible violation of integrity is prevented by *modifying* each user transaction that contains updates, such that the modified transaction cannot violate the integrity of the database.

A transaction is modified by extending it with additional extended relational algebra statements that implement the integrity control task for that transaction. The initial user transaction may require the modification with additional statements. As these statements may contain updates and may thus violate the integrity of the database themselves, a next addition of statements may be necessary. Consequently, transaction modification is a recursive algorithm as shown below.

Algorithm 5.1. Let T denote an arbitrary transaction and $\mathcal{J}$ a set of integrity rules defined on a database. Then the *modified transaction* of T with respect to $\mathcal{J}$ is defined by function $ModT$ as follows:

$$ModT(T, \mathcal{J}) = ModP(T\downarrow, \mathcal{J})\uparrow$$

where

$$ModP(P, \mathcal{J}) = \begin{cases} P & \text{if } TrigP(P, \mathcal{J}) = P_\epsilon \\ P \oplus ModP(TrigP(P, \mathcal{J}), \mathcal{J}) & \text{otherwise} \end{cases}$$

and

$$TrigP(P, \mathcal{J}) = TrOptRS(SelRS(P, \mathcal{J}))$$

The symbol $\downarrow$ denotes the *transaction debracketing* operator that converts a transaction into the relational algebra program by removing the transaction brackets. The *program bracketing* operator $\uparrow$ performs the inverse function: it adds transaction brackets to a relational algebra program. Finally, the *program concatenation* operator $\oplus$ concatenates two algebra programs.

In the above algorithm, function *SelRS* is used to select the integrity rules to be used to modify the transaction, and function *TrOptRS* for translating and optimizing integrity rules into efficient extended relational algebra programs that can be concatenated to the transaction.

Algorithm 5.2. The integrity rule selection is performed by function *SelRS*; this function selects a set of integrity rules from a set $\mathcal{J}$ based on the statements in extended relational algebra program P:

$$SelRS(P, \mathcal{J}) = \{J \in \mathcal{J} \mid triggers(J) \cap GetTrigP(P) \neq \emptyset\}$$

where

$$GetTrigP(P) = \begin{cases} \emptyset & \text{if } P = P_\epsilon \\ GetTrigS(head(P)) \cup & \\ \quad GetTrigP(tail(P)) & \text{otherwise} \end{cases}$$

and

$$GetTrigS(S) = \begin{cases} \{\langle INS, R\rangle\} & \text{if } S = insert(R, E) \\ \{\langle DEL, R\rangle\} & \text{if } S = delete(R, E) \\ \{\langle INS, R\rangle, \langle DEL, R\rangle\} & \text{if } S = update(R, E, \alpha) \\ \emptyset & \text{otherwise} \end{cases}$$

In the definition of *GetTrigS*, E denotes an arbitrary extended relational algebra expression of the same type as R, and α denotes an arbitrary update function.

5.2 Integrity Rule Optimization and Translation

To be able to perform transaction modification, integrity rules have to be translated into extended relational algebra. Before translation, rules are optimized. The optimization and translation of a set of rules is controlled by the function *TrOptRS*.

Algorithm 5.3. Given a set of integrity rules $\mathcal{J}$, function *TrOptRS* preprocesses these rules to produce a single extended relational algebra program:

$$TrOptRS(\mathcal{J}) = \begin{cases} P_\epsilon & \text{if } \mathcal{J} = \emptyset \\ TransR(OptR(head(\mathcal{J}))) \oplus & \\ \quad TrOptRS(tail(\mathcal{J})) & \text{otherwise} \end{cases}$$

Functions *OptR* and *TransR* deal with the optimization respectively translation of individual integrity rules. These functions are discussed in more detail below.

5.2.1 Integrity rule optimization

This section discusses the optimization of integrity rules, i.e. the transformation of a rule J into a rule J' that has the same semantics, but can be evaluated at a lower cost. Complete optimization of a rule consists of optimization

of the condition and the action of the rule. Here we focus on optimization of the condition. Optimization of relational algebra constructs is dealt with extensively in the field of query optimization (see e.g. [3]); techniques developed in this context can be used for the optimization of integrity rule actions.

Algorithm 5.4. Restricting the optimization of integrity rules to the optimization of the condition of the rule, the optimization function *OptR* can be defined as follows, where J is an integrity rule:

$$OptR(J) = \langle triggers(J), OptC(condition(J)), action(J) \rangle$$

The functionality of function *OptC* can be chosen freely within the boundaries of the equivalence criterium. A complete specification of this function is not within the scope of this paper. Techniques to be used in *OptC* can be the following [8, 10]: the use of differential relations to avoid unnecessary data access [18, 5, 7], syntactical manipulation of constraint specifications [14, 11], and semantic manipulation of constraint specifications [16].

5.2.2 Integrity rule translation

This section deals with the translation of integrity rules into extended relational algebra programs. In general, a program is derived from the condition and action of an integrity rule. We can distinguish between rules with an aborting violation response action and rules with a compensating action. If the rule has an aborting character, only the condition of the rule has to be translated to extended relational algebra constructs. So, we have the following.

Algorithm 5.5. Function *TransR* translates an integrity rule J into an extended relational algebra program:

$$TransR(J) = \begin{cases} TransC(condition(J)) \\ \quad \text{if } action(J) = abort \\ TransCA(condition(J), action(J)) \\ \quad \text{otherwise} \end{cases}$$

With respect to function *TransCA* some remarks can be made. In most practical cases, the specified violation response action exactly compensates all incorrect values in the database and has no other side effects. This implies that the program produced by function *TransCA* can be equal to the violation response action given as argument to the function. Other cases with side effects that do not depend on the rule condition require a deeper analysis of condition and action; this analysis is beyond the scope of this paper. Consequently, this section deals with function *TransC*, i.e. the translation of conditions in the $\mathcal{CL}$ language into the extended relational algebra.

The extended relational algebra introduced in Section 2 includes the constructs necessary for the construction of compensating programs. For the construction of aborting programs, however, a new construct has to be added. This construct is described and defined below.

Definition 5.1. Let E denote an arbitrary extended relational expression. Then the *alarm* statement $alarm(E)$ aborts the transaction it belongs to if E is non-empty; otherwise it has no effect:

$$alarm(E) \equiv \begin{cases} abort & \text{if } COUNT(E) > 0 \\ nothing & \text{otherwise} \end{cases}$$

Using the *alarm* statement, aborting constraint specifications can be translated to the relational algebra. As translation of relational calculus to relational algebra is dealt with extensively elsewhere (see e.g. [21, 15]), a complete translation algorithm is not presented here.

Constraints consisting of a quantified formula form an important class; for this class, function *TransC* can be defined as follows:

Algorithm 5.6. Let c denote a constraint (well-formed formula) in the $\mathcal{CL}$ language, and assume that c has the form $(\vartheta x)(c'(x))$, with $\vartheta \in \{\forall, \exists\}$. Then function *TransC* translates c into an aborting extended relational algebra construct:

$$TransC(c) = \begin{cases} alarm(CalcToAlg(\{x \mid \neg c'(x)\})) \\ \quad \text{if } c = (\forall x)(c'(x)) \\ alarm(\sigma_{attr_1=0} CNT(CalcToAlg(\{x \mid c'(x)\}))) \\ \quad \text{if } c = (\exists x)(c'(x)) \end{cases}$$

In the above algorithm, function *CalcToAlg* translates a relational tuple calculus expression in the equivalent relational algebra expression (see e.g. [21, 12, 15]).

The translation of a number of typical constructs in $\mathcal{CL}$ is shown in Table 1. In this table, c, c_1, and c_2 denote conditions defined on a tuple in $\mathcal{CL}$ format. The corresponding conditions c', c'_1, and c'_2 in the extended relational algebra are obtained by trivial syntactical modification. Further, $AGGR$ denotes an arbitrary aggregate function from $\mathcal{CL}$.

5.3 Trigger set generation

The selection of integrity rules in the transaction modification algorithm is based on the trigger sets of the rules. Although the trigger sets can be specified by the rule designer, it is more convenient and less error-prone if they are automatically generated by the database system.

C	$TransC(c)$
$(\forall x)(x \in R \Rightarrow c(x))$	$alarm(\sigma_{\neg c'} R)$
$(\forall x)(x \in R \Rightarrow (\exists y)(y \in S \wedge x.i = y.j))$	$alarm(\pi_i R - \pi_j S)$
$(\forall x)(x \in R \Rightarrow (\forall y)(y \in S \Rightarrow x.i \neq y.j))$	$alarm(\pi_i R \cap \pi_j S)$
$(\forall x,y)((x \in R \wedge y \in S \wedge c_1(x,y)) \Rightarrow c_2(x,y))$	$alarm(\sigma_{\neg c_2'}(R \bowtie_{c_1'} S))$
$(\exists x)(x \in R \wedge c(x))$	$alarm(\sigma_{attr_1=0}(CNT(\sigma_{c'} R)))$
$c(AGGR(R,i))$	$alarm(\sigma_{\neg c'}(AGGR(R,i)))$
$c(CNT(R))$	$alarm(\sigma_{\neg c'}(CNT(R)))$

Table 1: Translation of typical constraint constructs

The trigger set of an integrity rule can always be deduced automatically from the condition of the rule. The algorithm is based on the syntax of the $C\mathcal{L}$ language used for the specification of the condition (see Definitions 4.1-4.4).

Algorithm 5.7. A trigger set is generated from an integrity rule condition in the form of a $C\mathcal{L}$ well-formed formula W by function $GenTrigC$ as shown below. In this definition, the symbols V_u and V_e denote the set of universally respectively existentially quantified variables in a condition.

$$GenTrigC(W) = GenTrigW(W, \emptyset, \emptyset)$$

$$GenTrigW(W, V_u, V_e) =$$

$$\begin{cases} GenTrigW(W_1, V_u \cup \{x\}, V_e) & \text{if } W = (\forall x)W_1 \\ GenTrigW(W_1, V_u, V_e \cup \{x\}) & \text{if } W = (\exists x)W_1 \\ GenTrigW(W_1, V_u, V_e) \cup & \\ \quad GenTrigW(W_2, V_u, V_e) & \text{if } W = W_1 \phi W_2 \\ GenTrigN(W_1, V_u, V_e) \cup & \\ \quad GenTrigW(W_2, V_u, V_e) & \text{if } W = W_1 \Rightarrow W_2 \\ GenTrigN(W_1, V_u, V_e) & \text{if } W = \neg W_1 \\ GenTrigA(W, V_u, V_e) & \text{otherwise} \end{cases}$$

$$GenTrigN(W, V_u, V_e) =$$

$$\begin{cases} GenTrigN(W_1, V_u, V_e \cup \{x\}) & \text{if } W = (\forall x)W_1 \\ GenTrigN(W_1, V_u \cup \{x\}, V_e) & \text{if } W = (\exists x)W_1 \\ GenTrigN(W_1, V_u, V_e) \cup & \\ \quad GenTrigN(W_2, V_u, V_e) & \text{if } W = W_1 \phi W_2 \\ GenTrigW(W_1, V_u, V_e) \cup & \\ \quad GenTrigN(W_2, V_u, V_e) & \text{if } W = W_1 \Rightarrow W_2 \\ GenTrigW(W_1, V_u, V_e) & \text{if } W = \neg W_1 \\ GenTrigA(W, V_u, V_e) & \text{otherwise} \end{cases}$$

$$GenTrigA(A, V_u, V_e) =$$

$$\begin{cases} GenTrigT(T_1) \cup & \\ \quad GenTrigT(T_2) & \text{if } A = T_1 \vartheta T_2 \\ \{\langle INS, R\rangle\} & \text{if } A = (x \in R) \text{ and } x \in V_u \\ \{\langle DEL, R\rangle\} & \text{if } A = (x \in R) \text{ and } x \in V_e \\ \emptyset & \text{otherwise} \end{cases}$$

$$GenTrigT(T) =$$

$$\begin{cases} \{\langle INS, R\rangle, \langle DEL, R\rangle\} & \text{if } T = \Gamma_1(R,i) \\ \{\langle INS, R\rangle, \langle DEL, R\rangle\} & \text{if } T = \Gamma_2(R) \\ \emptyset & \text{otherwise} \end{cases}$$

where $\phi \in \{\wedge, \vee\}, \vartheta \in \{<, \leq, =, \neq, \geq, >\}$,
$\Gamma_1 \in \{SUM, AVG, MIN, MAX\}$
$\Gamma_2 \in \{CNT, MLT\}$

The function above completes the declarative specification of the integrity control technique using the transaction modification approach. Although the algorithms presented above can be used without modification, some improvements can be made that enhance their usability in real-world situations. These improvements are discussed in Section 6.

5.4 Example of transaction modification

To conclude the discussion of the basic transaction modification technique, a simple example is presented below.

Example 5.1. Take the simple beer database and the example integrity rules R_1 and R_2 described in Section 4. The trigger sets of these rules can be derived from the condition using the algorithm presented above. Now suppose that a user submits the following transaction adding a new tuple to the *beer* relation:

begin
insert(*beer*, ("*exportgold*", "*stout*", "*guineken*", 6));
end

Then the integrity control subsystem will modify this transaction to the following:

$$
\begin{array}{l}
begin \\
insert(beer, (\text{"}exportgold\text{"}, \text{"}stout\text{"}, \text{"}guineken\text{"}, 6)); \\
alarm(\sigma_{\neg(alcohol \geq 0)} beer); \\
temp = \pi_{brewery} beer - \pi_{name} brewery; \\
insert(brewery, \pi_{name,null,null} temp); \\
end
\end{array}
$$

The second statement implements the integrity control for the domain constraint, and the last two statements the integrity control for the referential integrity constraint. The modified transaction is now guaranteed to be correct and can be executed without any further precautions.

6 Operational aspects

The sections above have discussed the concepts behind the transaction modification technique. Some operational aspects require some more attention, however, to make the ideas more usable in practice. The most important aspects are discussed below.

6.1 Infinite triggering suppression

The fact that the transaction modification technique uses a static analysis to select the triggered integrity rules creates a relatively high risk of infinite triggering behaviour. The triggering behaviour of a set of integrity rules can be analyzed by means of a triggering graph as defined below.

Definition 6.1. Let $\mathcal{J}$ denote a set of integrity rules. Then the *triggering graph* of $\mathcal{J}$ is a directed graph $G = \langle V, E \rangle$. The set V denotes the vertices of G and corresponds with the set of integrity rules $\mathcal{J}$. The set E denotes the edges of G and is defined as follows:

$$
\begin{array}{rcl}
E & = & \{\langle J_1, J_2 \rangle \mid J_1, J_2 \in V \wedge \\
 & & GetTrigP(action(J_1)) \cap triggers(J_2) \neq \emptyset\}
\end{array}
$$

Infinite rule triggering in a rule set $\mathcal{J}$ can only occur if the triggering graph of $\mathcal{J}$ contains one or more cycles. A simple way to remove cycles is to allow actions of integrity rules to be declared as *non-triggering* extended relational algebra programs.

Definition 6.2. A *non-triggering relational algebra program* P_n will never trigger any rule. The trigger set derivation function *GetTrigP* as discussed before is replaced by function *GetTrigPX*:

$$
GetTrigPX(P) = \begin{cases} \emptyset & \text{if } non\text{-}triggering(P) \\ GetTrigP(P) & \text{otherwise} \end{cases}
$$

Given these concepts, an integrity control subsystem can validate a set of integrity rules with respect to triggering behaviour by constructing and analyzing the triggering graph. If cycles are detected, the system assists the user in removing these cycles. This approach is comparable to that described in [4]. It does, however, place a heavy burden on the shoulders of the database designer, since the system cannot guarantee integrity completely automatically in this approach.

The approach described above can be used to avoid 'unnecessary' infinite triggering behaviour. Of course it is possible to define a set of integrity rules that inherently implies an infinite triggering process. Clearly, such a set of rules has to be considered semantically incorrect.

6.2 Static optimization and translation

As specified by the definition of the transaction modification function *ModT*, integrity rules are optimized and translated each time a transaction is modified. Clearly, this is not necessary, as rules can be optimized and translated once when they are specified. The translated form is then stored for use at constraint enforcement time. This also requires the trigger set of a rule to be stored with a translated rule. Therefore, the concept of integrity program is defined below.

Definition 6.3. An *integrity program* defined on a database schema $\mathcal{D}$ is a pair $K = \langle t, p \rangle$ with the following components:

- The *trigger set* t is a set of pairs $\langle u, r \rangle$ with $u \in \{INS, DEL\}$ and r a relation name in $\mathcal{D}$.
- The *triggered program* p is an extended relational algebra program.

If $K = \langle t, p \rangle$ denotes an integrity program, $triggers(K)$ denotes the trigger set t, and $action(K)$ denotes the triggered program p.

This definition of an integrity program can easily be extended with a flag indicating whether the program is non-triggering as discussed above.

An integrity program is derived from an integrity rule J using the rule optimization and translation algorithms presented in Section 5:

Algorithm 6.1. The integrity program generation function *GetIntP* is defined as follows, where J is an integrity rule:

$$GetIntP(J) = \langle triggers(J), TransR(OptR(J)) \rangle$$

Given the fact that integrity rules are translated at rule definition time and stored in a set of integrity programs, the transaction modification algorithm has to be adapted. The changes to the algorithm are rather straightforward:

Algorithm 6.2. Function *TrigP* from the transaction modification algorithm is redefined as follows. Here P is an extended relational algebra program and $\mathcal{K}$ a set of integrity programs.

$$TrigP(P, \mathcal{K}) = ConcatP(SelPS(P, \mathcal{K}))$$

where

$$SelPS(P, \mathcal{K}) = \{K \in \mathcal{K} \mid triggers(K) \cap GetTrigP(P) \neq \emptyset\}$$

$$ConcatP(\mathcal{K}) = \begin{cases} P_\epsilon & \text{if } \mathcal{K} = \langle\rangle \\ action(head(\mathcal{K})) \oplus ConcatP(tail(\mathcal{K})) & \text{otherwise} \end{cases}$$

As may be clear, in the above algorithm function *SelPS* is the substitute for function *SelRS*, and *ConCatP* is the substitute for *TrOptRS*. Note that rule optimization and translation are not included in *ConCatP*, since these tasks are not performed at constraint enforcement time. Note further that the definition above interprets the set $\mathcal{K}$ as a list; a set can be interpreted as a list by imposing an arbitrary order on the elements of the set.

7 Conclusions

This paper shows that it is possible to give a precise declarative specification of a complete integrity control subsystem. The specification is based on the the transaction modification technique, which has two important characteristics. In the first place, transaction modification has a well-defined formal background as presented in the first sections of this paper. This provides a good basis for discussing the precise semantics of integrity control, something that is lacking in many system-oriented proposals. In the second place, the specification can be directly used for the design and implementation of an integrity control subsystem that can be used in real-world practice. In contrast to many theory-oriented proposals, the transaction modification approach deals with constraint enforcement efficiency and multi-update transactions. The technique can easily be mapped to an abstract DBMS system architecture [8].

The transaction modification algorithms as presented in this paper can be easily extended in a number of directions. An extension to a multi-set relational algebra is presented in [8]. As a multi-set algebra is closely connected to SQL-like environments, this can be an important factor in the usability of the technique in practice. An extension to a parallel database system environment with fragmented relations is presented in [7, 8]. This extension clearly demonstrates, that transaction modification is well fit for high-performance database environments. Further, transaction modification can be used for purposes other than integrity control as well, like materialized view maintenance [8].

The feasibility of the technique in practice is demonstrated with a prototype transaction modification subsystem in the parallel main-memory PRISMA/DB database system [1]. A performance evaluation on this system has shown the high performance that can be obtained by integrity control through transaction modification. Details on the performance evaluation can be found in [8, 9]. The performance of the prototype system is merely illustrated here with the following. Given a test database with a key relation of 5000 tuples and a foreign key relation of 50000 tuples, checking a referential integrity constraint after the insertion of 5000 new tuples into the foreign key relation can be completed within 3 seconds on an 8-node POOMA multiprocessor [22]. Checking a domain constraint in the same situation takes less than 1 second. One may conclude that the evaluation clearly demonstrates that constraint enforcement costs do not have to be an obstacle for integrity control in practice.

Acknowledgements

Thanks go to Jennifer Widom from IBM Almaden Research Center at San Jose, USA, and Peter Apers and Rolf de By from the University of Twente for their comments and suggestions for improvement with respect to earlier versions of the work presented in this paper. Hennie Steenhagen from the University of Twente is thanked for her comments on the draft of this paper.

References

[1] P.M.G. Apers, C.A. v.d. Berg, J. Flokstra, P.W.P.J. Grefen, M.L. Kersten, A.N. Wilschut; *PRISMA/DB: A Parallel, Main-Memory Relational DBMS*; IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6, 1992.

[2] P. Bernstein, B. Blaustein, E. Clarke; *Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data*; Procs. 6th Int. Conference on Very Large Data Bases; Montreal, Canada, 1980.

[3] S. Ceri, G. Pelagatti; *Distributed Databases, Principles and Systems*; McGraw-Hill, New York, USA, 1984.

[4] S. Ceri, J. Widom; *Deriving Production Rules for Constraint Maintenance*; Procs. 16th Int. Conference on Very Large Data Bases; Brisbane, Australia, 1990.

[5] C. Gardarin, P. Valduriez; *Relational Databases and Knowledge Bases*; Addison-Wesley, Reading, USA, 1989.

[6] J. Gray; *The Transaction Concept: Virtues and Limitations*; Procs. 7th Int. Conference on Very Large Data Bases; Cannes, France, 1981.

[7] P.W.P.J. Grefen, P.M.G. Apers; *Parallel Handling of Integrity Constraints on Fragmented Relations*; Procs. Int. Symposium on Databases in Parallel and Distributed Systems; Dublin, Ireland, 1990.

[8] P.W.P.J. Grefen; *Integrity Control in Parallel Database Systems*; Ph.D. Thesis, University of Twente, 1992.

[9] P.W.P.J. Grefen, J. Flokstra, P.M.G. Apers; *Performance Evaluation of Constraint Enforcement in a Parallel Main-Memory Database System*; Procs. 3rd Int. Conference on Database and Expert System Applications; Valencia, Spain, 1992.

[10] P.W.P.J. Grefen, P.M.G. Apers; *Integrity Control in Relational Database Systems - An Overview*; Journal of Data and Knowledge Engineering, Vol.10, No.2, 1993; North Holland - Elsevier.

[11] A. Hsu, T. Imielinsky; *Integrity Checking for Multiple Updates*; Proceedings of the 1985 ACM SIGMOD International Conference on the Management of Data; Austin, USA, 1985.

[12] A. Klug; *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions*; Journal of the ACM, vol. 29, No. 3, 1982.

[13] H.F. Korth, A. Silberschatz; *Database System Concepts*; McGraw-Hill, New York, USA, 1986.

[14] J.M. Nicolas; *Logic for Improving Integrity Checking in Relational Data Bases*; Acta Informatica 18, 1982.

[15] J. Paredaens, P. de Bra, M. Gyssens, D. van Gucht; *The Structure of the Relational Database Model*; Springer-Verlag, Berlin, Germany, 1989.

[16] X. Qian, G. Wiederhold; *Knowledge-based Integrity Constraint Validation*; Proceedings of the 12th International Conference on Very Large Data Bases; Kyoto, Japan, 1986.

[17] E. Simon, P. Valduriez; *Design and Implementation of an Extendible Integrity Subsystem*; Procs. 1984 ACM SIGMOD Int. Conference on the Management of Data; Boston, USA, 1984.

[18] E. Simon, P. Valduriez; *Design and Analysis of a Relational Integrity Subsystem*; MCC Technical Report DB-015-87; MCC, Austin, USA, 1987.

[19] M. Stonebraker; *Implementation of Integrity Constraints and Views by Query Modification*; Procs. 1975 ACM SIGMOD Int. Conference on the Management of Data; San Jose, USA, 1975.

[20] M. Stonebraker, G. Kemnitz; *The POSTGRES Next-Generation Database Management System*; Communications of the ACM, Vol.34, No.10, 1991.

[21] J.D. Ullman; *Principles of Database Systems, Second Edition*; Computer Science Press, Rockville, USA, 1982.

[22] M.C. Vlot; *The POOMA Architecture*; Procs. of the PRISMA Workshop on Parallel Database Systems; Noordwijk, The Netherlands, 1990.